SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning

Jianlei Chi, Yu Qu, Ting Liu, *Member, IEEE,* Qinghua Zheng, *Member, IEEE,* Heng Yin, *Member, IEEE*

Abstract—Software vulnerabilities are now reported at an unprecedented speed due to the recent development of automated vulnerability hunting tools. However, fixing vulnerabilities still mainly depends on programmers' manual efforts. Developers need to deeply understand the vulnerability and try to affect the system's functions as little as possible.
In this paper, with the advancement of Neural Machine Translation (NMT) techniques, we provide a novel approach called SeqTrans to exploit historical vulnerability fixes to provide suggestions and automatically fix the source code. To capture the contextual information around the vulnerable code, we propose to leverage data flow dependencies to construct code sequences and fed them into the state-of-the-art transformer model. Attention and copy mechanisms are both exploited in SeqTrans. We evaluate SeqTrans on both single line and multiple line vulnerability fixes on a dataset containing 1,282 commits that fix 624 vulnerabilities in 205 Java projects. Results show that the accuracy of SeqTrans can achieve 77.6% in single line fix and 52.5% in multiple line fix. In the meantime, we look deep inside the result and observe that NMT model performs very well in certain kinds of vulnerabilities like CWE-287 (Improper Authentication) and CWE-863 (Incorrect Authorization).

Index Terms—Software vulnerability, vulnerability fix, neural machine translation, machine learning

1 Introduction

Software evolves quite frequently due to numerous reasons such as deprecating old features, adding new features, refactoring, vulnerability fixing, etc. Software vulnerability is one of the major threats to software security. Vulnerabilities like HeartBleed [1], Spectre [2] and Meltdown [3], introduced significant threats to millions of users. Vulnerabilities are reported at an unprecedented speed due to the recent development of automated vulnerability hunting tools like AFL [4], AFLGo [5], AFLFast [6]. On the other hand, fixing vulnerabilities still mainly depends on programmers' manual efforts, which are tedious and error-prone. Automatically learn to generate vulnerability fixes is urgently needed and will greatly improve the efficiency of software development and maintenance processes.

There are a large number of works of automated program repair or called code migration in both industrial and academic domains. [7]. Many research works focus on one type of code modification, such as API change [8], [9], [10], [11], [12] and suggestion [13], refactoring [14], [15]. IDEs also provide specific kinds of automatic changes [16]. For example, refactoring, generating getters and setters, adding override/implement methods, etc. However, although some promising results have been achieved, current works of automated program repair face a list of limitations. Firstly, most of them heavily rely on domain-specific knowledge or predefined change templates, which leads to limited scalability [7]. Secondly, traditional techniques leverage search space, statistical analysis to rank similar repair records needs to define numerous features, which can be time-consuming and not accurate enough. In this paper, we focus on automatic vulnerability fixing that relies entirely on machine learning to capture grammatical and structural information as common change patterns. By combining vulnerability fixing with machine learning, our goal is to assist the developer in getting rid of tedious repair works and benefiting from training with the continuous growing historical vulnerability fixing records.

To model these historical records, we choose the general framework of Neural Machine Translation (NMT) to learn rules from historical records and apply them in future edits. It is widely utilized in Natural Language Processing (NLP) domain, such as translate one language (e.g., English) to another language (e.g., Swedish). NMT model can generalize numerous sequence pairs between two languages and learn the probability distribution of changes, assign higher weights to appropriate editing operations. Previous works such as Tufano et al. [17] and Chen et al. [18] have shown an initial success of using the NMT model for predicting code changes. However, both of them only focus on simple scenarios such as short sequences and single line cases. In fact, since the NMT model is originally exploited for natural language, we should think about the gap between natural language and programming language [19]. Firstly, program language falls under the category of languages called context-sensitive languages. Dependencies in one statement may come from the entire function or even the entire class. Nevertheless, in natural language, token dependencies are always distributed in the same sentence or neighbouring sentences. Secondly, the vocabulary of natural languages is filled with conceptual terms. The vocabulary of programming languages is generally only grammar word like essential comments, plus various custom-named things like variables and functions. Thirdly, programming languages are unambiguous, while natural languages are often multiplied ambiguous and require interpretation in context to be fully understood.

In this work, in order to solve the dependency problem across the entire class, we construct the define-use (def-use) [20] chain which represents the data flow dependencies to capture important context around the vulnerable statement.

Another problem previous works do not mention is that they only focus on single line prediction, which means only statement replacement is supported. If one vulnerability fixing contains statement insertion or deletion, these works will fail to change the code or even ignore it. In this case, we also try to construct def-use chains for multi-line fixing to cover statement deletion and addition. Last but not the least, the seq2seq model [21] that previous works used cannot process long sentences very well. Therefore the token numbers are limited to 1000 [18] or even 100 [17]. To solve this problem, we introduce the state-of-the-art transformer model [22] to reduce the performance degradation caused by long statements. This enables us to process long statements and captures a broader range of dependencies.

We called our approach SeqTrans, and it works as follows: Firstly, we collect historical vulnerability fixing records, carefully create statement-level and function-level training and testing set for single line and multiple line prediction tasks. Secondly, we leverage a transformer model with attention and copy mechanism [23] to address existing problems mentioned before. Thirdly, if new vulnerable object is inputted to the trained model, beam search will be utilized first to obtain a list of candidate predictions. Then, a syntax checker will be used to check the list and select the most suitable candidate prediction. Recovered patching will be generated to developers. In order to evaluate our approach, we calculate the accuracy of both single line and multiple line predictions over 624 publicly disclosed vulnerabilities affecting 205 distinct open-source Java projects from the work of Ponta et al. in MSR 2019 [24]. The experimental result shows that our approach SeqTrans reaches a promising accuracy of single line prediction by 77.6%, outperforms the state-of-the-art model SequenceR [18] by 17.9% and substantially surpasses the performance Tufano et al. [17] and other NMT models. As for multiple line prediction, our approach also achieves the accuracy of 52.5%. To the best of our knowledge, this is the first report on utilizing sequence-to-sequence learning for multi-line prediction on vulnerability fix.

In the meantime, we also observed internally what types of vulnerability fixes can be well predicted by SeqTrans. An interesting observation we find is that our model has a large gap between different types of CWEs. Our model performs quite well in specific types of CWEs like CWE-287 (Improper Authentication) and CWE-863 (Incorrect Authorization) but even cannot make any prediction for certain CWEs like CWE-918 (Server-Side Request Forgery). We conclude training a general model to fix vulnerabilities automatically is too ambitious to cover all cases. But if we can focus on specific types of them, NMT model can make a very promising result to help developers. SeqTrans can actually cover about 60% of the types of CWEs in the data set.

The paper makes the following contributions:

1) We introduce the NMT model transformer to learn and generalize common patterns from data for vulnerability fixing.
2) We propose to leverage data flow dependencies to construct vulnerable sequences and maintain the vital context around them.
3) We implement our approach SeqTrans and evaluate 624 real publicly disclosed vulnerabilities affecting 205 distinct open-source Java projects. Our SeqTrans outperforms other program repair technique and is able to achieve the accuracy of 73.6% in single line prediction and 52.5% in multiple line prediction.
4) We make an internal observation about prediction results on different CWEs and find some interesting CWE fixing operations captured by our model. Our model can predict specific types of CWEs pretty well.

2 Motivation Example

Figure 1 shows a motivating example of our approach. In Figure 1, there are two vulnerability fixes for CVE-2017-1000390 and CVE-2017-1000388, respectively. These two CVEs belong to the same CWE: CWE-732, which is named "Incorrect Permission Assignment for Critical Resource." CWE-732 emphasizes that "the product specifies permissions for a security-critical resource in a way that allows that resource to be read or modified by unintended actors," which means that when using a critical resource such as a configuration file, the program should carefully check if the resource has insecure permissions.

In Figure 1 (a), before the function `getIconFileName` returns the $IconFileName$, it should check whether the user has the corresponding permission. In Figure 1 (b), before the function `EdgeOperation` accesses two resources $JobName$, it should also check the permission.

Although these two CVEs belong to different projects, their repair processes are very similar. This inspired us that it might be possible to learn common patterns from historical vulnerability fixes that correspond to the same or similar CWEs.

In this paper, we propose a novel method to exploit historical vulnerability fix records to provide suggestions and automatically fix the source code. If the function with similar structure requests accesses to a critical resource, our deep learning model can learn to check permissions before allowing access, eliminating the tedious process for developers to search for vulnerability and recapitulate repair patterns.

3 Methodology

We introduce the neural machine translation method to guide automatically vulnerability fixing, which aims at learning common change patterns from historical records and applying them on the new input files. The overview of our approach is given in Figure 2, which contains two stages: Preprocessing and training, prediction and patching.

SeqTrans provides fixed predictions at two granularity: statement level and method level, due to the reason that statement level (single line) prediction itself is not enough since it only considers code replacement. Another reason is that SeqTrans can work with other vulnerability detection tools such as Eclipse Steady [26]. They always provide vulnerability location information at the method level. Then, we perform normalization and abstraction based on data flow dependencies to extract the def-use chains. We believe def-use chains are suitable for deep learning models to

```
26  27      public String getIconFileName() {
27      -           return "plugin/jenkins-multijob-plugin/tool32.png";
        28  +       return Jenkins.getInstance().hasPermission(Jenkins.ADMINISTER) ? "plugin/jenkins-multijob-plugin/tool32.png" : null;
28  29      }
```

(a) CVE-2017-1000390, jenkinsci/tikal-multijob-plugin, 2424cec7a099fe4392f052a754fadc28de9f8d86

```
35  35      public EdgeOperation(String sourceJobName, String targetJobName) {
36  36          this.source = Jenkins.getInstance().getItemByFullName(sourceJobName.trim(), AbstractProject.class);
37  37          this.target = Jenkins.getInstance().getItemByFullName(targetJobName, AbstractProject.class);
        38  +       source.checkPermission(Permission.CONFIGURE);
        39  +       target.checkPermission(Permission.CONFIGURE);
```

(b) CVE-2017-1000388, jenkinsci/tikal-multijob-plugin, d442ff671965c279770b28e37dc63a6ab73c0f0e

Fig. 1: Two similar vulnerability fixes belonging to CWE-732

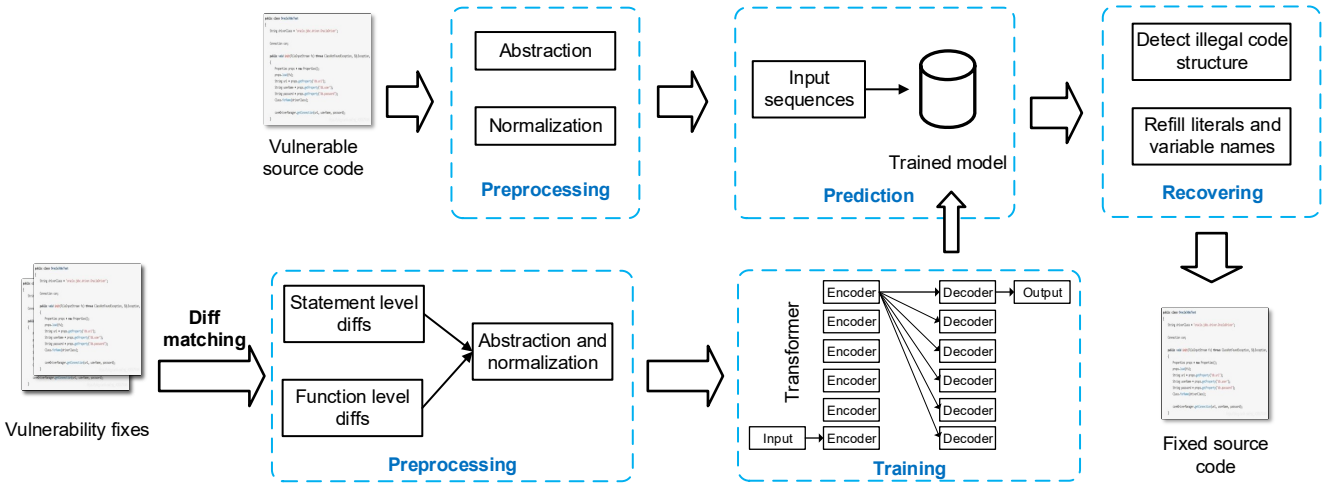


Fig. 2: Overview of our SeqTrans for automatically vulnerability fixing

capture syntax and structure information around the vulnerabilities with fewer noises. Then, these def-use chains are fed into the trained transformer model to generate a list of candidate predictions automatically. Syntax checker is exploited to check the error and select the best prediction (or predictions). After that, we refill abstraction and generate patches. We will discuss the details of each part in the following part of this section.

3.1 Code Change Mining

The dataset we exploit [1] [24] provides vulnerability fixing records as following shows:

$$(vulnerability\_id; repository\_url; commit\_id)$$

where $vulnerability\_id$ is the identifier of a vulnerability that is fixed in the $commit\_id$ in the open source code repository at the $repository\_url$. Each line in the dataset represents a commit that contributes to fixing a vulnerability. Then, we utilize a crawler to collect program repositories mentioned in the dataset. Pull Request (PR) data will be extracted based on $commit\_id$. After that, in each PR we need to find out java file changes involved. Because our approach SeqTrans only supports java files now. With the help of a git version control system JGit [27], we can retrieve the version of java files before and after code changes implemented in the PR. We call these java file pairs $ChangePair(CP)$, each CP contains a list of code diffs.

1. https://github.com/SAP/vulnerability-assessment-kb

3.2 Code Diff Extraction

After we obtaining CPs from PR, we need to locate the method-level code changes and statement-level codes changes for multi-line prediction and single line prediction. Although we can exploit the "git diff" command provided by git to search line-level code diffs, they are not precise enough. Even a slight code structure change such as a new-line, adding space will be recognized as a code diff. For this reason, we choose to search for code diffs by using Abstract Syntax Trees (ASTs). The state-of-the-art diff searching tool named GumTree [28] is utilized to search for fine-grained AST node mappings. It is worth noting that GumTree only provides a fine-grained mapping between AST nodes, so we modified the code of GumTree and combined with another tool, Understand [29], to extract the method-level and statement-level code diffs. In the meantime, we found some bugs of Gumtree that leads to incorrect mismatching and reported to the author. After that, each CP is represented as a list of code diffs:

$$CP = (m_{src}, m_{dst})_1, ..., (m_{src}, m_{dst})_n$$

$$CP = (st_{src}, st_{dst})_1, ..., (st_{src}, st_{dst})_n$$

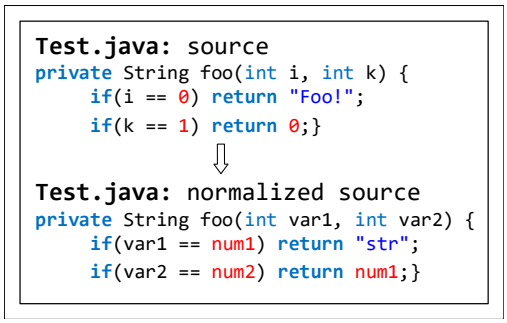


Fig. 3: Normalize the source code

where (m_{src}, m_{dst}) represents method from the source file and the destination file. Similarly, the statement-level code diff will be represented as (st_{src}, st_{dst})

Then, we will extract data flow dependencies around code diffs to construct our def-use chains. The reasons why we use data flow dependencies are shown as follows: 1) Context around the vulnerable statements is valuable to understand the risky behavior and capture structure relationships. However, it is too heavy to maintain the full context in the class-level with lots of unrelated code. 2) Data flow dependencies provide enough context for transformation. If one use statement needs to be modified, there is a high probability to co-change its definition statements simultaneously. 3) Control flow dependencies often contain branches, which makes them too long to be tokenized.

The definition and use (def-use) dependencies can be extracted from the ASTs. The process can be shown as follows: Firstly, we traverse the whole AST and label each variable name. These variable names are distributed over the leaf nodes of the AST. Then, We will traverse up the leaf node to its defined location. With the help of modified GumTree and Understand, SeqTrans changes each CP as the following shows:

$$CP = ((def_1, ..., def_n, m_{src}), (def_1, ..., def_n, m_{dst}))_1, ..., ((def_1, ..., def_n, m_{src}), (def_1, ..., def_n, m_{dst}))_n$$

$$CP = ((def_1, ..., def_n, st_{src}), (def_1, ..., def_n, st_{dst}))_1, ..., ((def_1, ..., def_n, st_{src}), (def_1, ..., def_n, st_{dst}))_n$$

In this paper, we ignore code changes that involve the addition or deletion of entire methods/files.

3.3 Code Abstraction & Normalization

In the training process of the NMT model, there exist a couple of drawbacks. Because NMT models output a probability distribution over words, they can become very slow with a large number of possible words. We need to impose an artificial limit on how of the most common words we want our model to handle. This is also called the vocabulary size. In order to reduce the vocabulary size, we need to preserve the semantic information of the source code while abstracting the context.

The normalization process is shown in Figure 3. We replace variable names to "$var1$", ..., "$varn$", each literal and string are also replaced to "$num1$", ..., "$numn$" and "str". The reasons why we do this involves: 1) reduce the vocabulary size and the frequency of specific tokens; 2) reduce the redundancy of the data and improve the consistency of the data. We will maintain a dictionary to store the mappings between the original label and the substitute so that they can be refilled after prediction. Through the above optimization, we can control the vocabulary size to about 1500, which makes the NMT model to concentrate on learning common patterns from different code changes.

Subsequently, we split each abstract CP into a series of tokens. It is worth to mention that the seq2seq model utilized in previous works faces severe performance degradation when processing long sequences. For example, Tufano et al.[17] limited the token number to 50-100, Chen et al.[18] limited the token number to 1000. Because the transformer model we utilized can better handle long sequences. In our approach, we will limit the statement-level CP to 1500 tokens and not limit the length of function-level CP. We will discuss the details in the following subsection.

3.4 Neural Machine Translation Network

In this phase we train SeqTrans to learn how to transform the vulnerable codes to correct version and generate patches.

3.4.1 *Transformer Model*

In this work, we choose to use the transformer model [22] to solve the performance degradation problem of the seq2seq model on long sequences. It has been widely utilized by OpenAI and DeepMind in their language models. Unlike Recurrent Neural Network (RNN) [30] or Long Short Term Memory (LSTM) [31] models, transformer relies entirely on the attention mechanism to draw global dependencies between input and output data. This model is more parallel and achieves better translation results. The transformer consists of two main components: a set of encoders chained together and a set of decoders chained together. The encode-decoder structure is widely used in NMT models, the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to an embedding representation $z = (z_1, ..., z_n)$, which contains information about the parts of the inputs which are relevant to each other. Given z, the decoder then exploits this incorporated contextual information to generate an output sequence. Generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model consumes the previously generated symbols as additional input when generating the next [32]. The transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder. Each encoder and decoder make use of an attention mechanism to weigh the connections between every input and refer to that information to generate output [22].

As for the parameter selection, we discussed a variety of settings for SeqTrans. Most of the major components are verified with the sensitivity analysis experiments in RQ3. The model is trained with a batch size of 4096 for 30000 iterations. In order to prevent the overfitting problem, we use a dropout of 0.1. In relation to the components shown in RQ3, some primary parameters are shown as follows:

- Word vector size : 512
- Attention layers: 6

- Size of hidden transformer feed-forward: 2048
- Dropout:0.1
- Batch size: 4096
- Train steps: 30000

3.4.2 Encoder

The encoder is composed of a stack of 6 identical layers. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. Residual connection [33] and normalization [34] have been employed to each sub-layer so that we can represent the output of the sub-layer as:

$$sub\_layer\_output = Layer\_normization(x+(SubLayer(x)))$$

where $Sublayer(x)$ is the function implemented by the sub-layer itself. The self-attention mechanism takes in a set of input encodings from the previous encoder and weighs their relevance to each other to generate a set of output encodings. The feed-forward neural network then further processes each output encoding individually. These output encodings are finally passed to the next encoder as its input. All sub-layers as well as the embedding layers produce outputs of dimension $d_{model} = 512$

3.4.3 Decoder

The decoder also contains a stack of 6 identical layers. However, each layer consists of three sub-layers: an attention sub-layer has been added to perform multi-head attention to draw relevant information from the encodings generated by the encoders. A masking has been used to prevent positions from attending to subsequent positions and ensure that the predictions for position i can depend only on the known outputs at positions less than i [22]. The other parts are the same as the encoder.

3.4.4 Attention Mechanism

The purpose of an attention mechanism is to use a set of encodings to incorporate context into a sequence. For each token the attention mechanism requires a query vector Q of dimension d_k, a key vector K of dimension d_k and a value vector V of dimension d_v. These vectors are created by multiplying the embedding by three matrices that we trained during the training process. Self-attention refers to the situation where the queries, keys, and values are all created using encodings of the sequence. Then the output Z of this attention mechanism is:

$$Z = Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{n}})V$$

The multi-head attention utilized in transformer implements several attention mechanisms in parallel and then combine the resulting encoding in a process.

3.4.5 Beam Search

In many cases, developers have certain domain-specific knowledge. We can generate a list of prediction results to let them pick the most suitable one. Instead of greedily choosing the most likely next step as the sequence is constructed, the beam search [35], [36] expands all possible next steps and keeps the k most likely, where k is a user-specified parameter and controls the number of beams or parallel searches through the sequence of probabilities. These k candidates will be provided as suggestions to developers to select the best result.

3.5 Patch Generation

The original output (or a list of outputs) is far from the version that can be successfully compiled. Because it contains abstraction and normalization, it even may contain grammatical errors after prediction. Our patch generation consists of two steps to solve these problems: abstraction refill and grammar check.

3.5.1 Abstraction Refill

As mentioned above, we maintain a dictionary to store the necessary information for restoration before abstraction. After prediction, the output will be concretized and all the abstraction contains in the dictionary will be refilled. The code will be automatically indented in this process. It should be noted that all comments will be deleted and will not be refilled again.

3.5.2 Grammar Check

We combine beam search with a grammar check tool to analyze the syntax and grammatical errors contained in the predictions. The static analysis tool *FindBugs* [37] is exploited to identify different types of potential errors in Java programs. Potential errors can be divided into four levels: scariest, scary, troubling, and of concern based on their possible impact or severity. In our SeqTrans, if the top 1 candidate prediction cannot pass the check of FindBugs and contains scariest or scary level bugs, we will search for the candidate list provided by beam search to test the next candidate until anyone has passed the check process. Finally, we can generate the newly patched file and provide it to developers.

We provide flexible choices for developers whether to enable this feature or judge by their domain-specific knowledge. In addition, we believe that with the continuous improvement of model training, these grammatical errors will become less and less. In the end, we will no longer rely on third-party grammatical error check tools.

4 Empirical Study & Evaluation

In this section, we conduct our experiment on a public dataset [24] of vulnerability fixes and evaluate our method: SeqTrans by investigating two research questions.

4.1 Research Questions

We explore the following research questions:

- **RQ1:** Can SeqTrans be competent for neural machine learning to capture common features and complete predictions?
  RQ1 aims to prove that NMT is a feasible approach to learn code transformations and outperforms other state-of-the-art techniques.
- **RQ2:** Does SeqTrans perform better in predicting specific types of CWEs?

RQ2 will explore in depth the prediction results and the source codes of the data set to observe whether our method performs inconsistently when predicting different kinds of code transformations.

- **RQ3:** Sensitivity analysis of SeqTrans.
  RQ3 will evaluate the impacts of the main components of SeqTrans on performance such as the data structure and the transformer model.

4.2 Experimental Design

In this section, we discuss our experimental design for RQ1, RQ2 and RQ3.

Dataset: Our evaluation is based on a public vulnerability repair dataset in MSR 19 [24]. The data was obtained both from the National Vulnerability Database (NVD) and from project-specific Web resources that they monitor on a continuous basis. From that data, they extracted a dataset that maps 624 publicly disclosed vulnerabilities affecting 205 distinct open-source Java projects, used in SAP products or internal tools, onto the 1282 commits that fix them. Out of 624 vulnerabilities, 29 do not have a CVE identifier at all and 46, which do have a CVE identifier assigned by a numbering authority, are not available in the NVD yet. These vulnerabilities have been removed from the dataset, the final number of vulnerabilities is 549.

The dataset is released under an open-source license, together with supporting scripts that allow researchers to automatically retrieve the actual content of the commits from the corresponding repositories and to augment the attributes available for each instance. Also, these scripts allow to complement the dataset with additional instances that are not security fixes (which is useful, for example, in machine learning applications).

We choose two different deduplication strategies: the first one is to remove all duplicates; The second strategy is to delete duplicate data between each commit. We believe that the second strategy can better simulate the data scenarios in the real environment, but we found that other techniques generally use the first strategy. We call them D_{small} and D_{median}

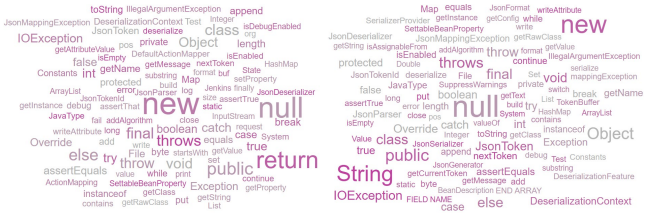
(a) D_{small} for single line prediction (b) D_{median} for single line prediction

Fig. 4: Data distribution for single line prediction

Figure 4 and Figure 5 shows the toke distribution of each dataset. We can find that different deduplication strategies will produce different token distributions.

4.2.1 *RQ1 Setup:*

We discuss our design for RQ1 from model used, comparison, and metric used, which are illustrated as follows:



(a) D_{small} for multiple line prediction (b) D_{median} for multiple line prediction

Fig. 5: Data distribution for multiple line prediction

Model used: In order to evaluate the performance of SeqTrans, we implement and train two different NMT models: Seq2seq, and Transformer model.

Seq2seq model is a RNN encoder-decoder model which has been widely utilized in NMT domain, previous works such as SequenceR [18] and Tufano et al. [17] are also based on this model. Transformer model has been introduced in the previous section.

Metric used: We have calculated the prediction accuracy for each technique. Prediction accuracy will be calculated using 10 cross validation for each technique. Then we will calculate the number of correct predictions divided by the total number to calculate the accuracy.

Comparison: Given the same dataset, we extract function-level and statement-level code diffs with def-use chains to separately train two models (seq2seq and transformer) for single line and multiple line predictions.

In single line prediction, we will compare the transformer model with and Tufano [17], [38] et al. and SequenceR [18]. Tufano has investigated the feasibility of using neural machine translation for learning wild code. The disadvantage of his method is that only sentences with less than 100 tokens are analyzed. SequenceR presents a novel end-to-end approach to program repair based on sequence-to-sequence learning. It utilizes the copy mechanism to overcome the unlimited vocabulary problem. To the best of our knowledge, it achieves the best result reported on such a task. However, the abstract data structure of this method retains too much useless context. It does not use the normalization method either. We have also added the model that utilizing the same data structure as we but using seq2seq model.

In multiple line prediction, we will just compare the transformer model with seq2seq model. To the best of our knowledge, we have not seen related works focus on multi-line prediction.

4.2.2 *RQ2 Setup:*

In this part, we will discuss the observations when we look deep inside the prediction result. We only manually analyze the prediction results generated by SeqTrans in single line prediction. Other models are not considered.

Metric used: We have calculated the prediction accuracy for each CWE and each category of code transformation.

Comparison: We will compare the accuracy for each CWE and each category of code transformation between the two dataset D_{small} and D_{median}. If some CWEs perform very differently between the two data sets, we will conduct a detailed case study.

4.2.3 ***RQ3 Setup***:

In this part, we will discuss the impacts of the main factors that affect the performance of SeqTrans.

The process is shown as follows: Firstly, we will select a list of parameters that may affect the performance of our model. Then we will change one parameter in one time and make the experiment in the same dataset. The final parameter selections of SeqTrans will produce the highest acceptance rates for the configurations we tested. For each parameter, we will utilize cross validation for 10 times and calculate the mean value as the final precision. The training set we choose is D_{median} for single line prediction.

Metric used: Parameters that we have tested contain encoder and decoder layers, word size of the embedding, data structure, code normalization and copy mechanism.

4.3 Experimental Results

4.3.1 RQ1: Can SeqTrans be competent for neural machine learning to capture common features and complete predictions?

Table 1 shows the accuracy results of single line prediction in four different NMT models including the transformer model that we exploit, Seq2Seq model, SequenceR and the work of Tufano et al.. For Seq2Seq model and transformer model, we use the same training set with def-use chains. As for the SequenceR [18] and Tufano et al. [38], we strictly follow their original codes and data structures. We have tried to exploit the beam search to generate a list of predictions. Figure 6 shows the performance on D_{small} when beam size increases from 1 to 50. Figure 7 shows the performance on D_{median} when beam size increases from 1 to 50. The x-axis represents beam size and the y-axis represents the prediction accuracy. For example, if Beam=10, for each input we will search all 10 generated predictions. When one of the prediction results is identical to the code transformation performed by developers, we determine that it is a correct prediction. We employ 10 cross-validation to calculate the accuracy of the model, in which 90% is utilized as a training set and validate set and the remaining 10% is used as a test set. If the predicted statement equals to the statement in the test set, there is a right prediction.

From the table, we see that our SeqTrans performs the best and achieves an accuracy of 242/2130 (11.3%) when Beam=1 on D_{small}, followed by Seq2seq 121/2130 (7.5%), SequenceR 252/3661 (6.9%) and Tufano et al. 20/1010 (2.0%). On D_{median}, SeqTrans also reaches the best accuracy of 2052/3334(61.6%), followed by SequenceR 2498/4610 (54.2%), Seq2seq 1621/3334 (48.6%) and Tufano et al. 252/1577 (15.9%).

To our surprise is that SequenceR is not as good as described. It even performs worse than Seq2seq when beam=1 on D_{small}. The poor performance of SequenceR can be explained by the difference between data structures. SequenceR utilize the buggy context which contains the buggy line and the context around the buggy line in the same function. Other variable declarations and method declarations in the same class will be retained, too. However, this buggy context keeps a lot of statements that have no relationship with the buggy line. The whole data structure

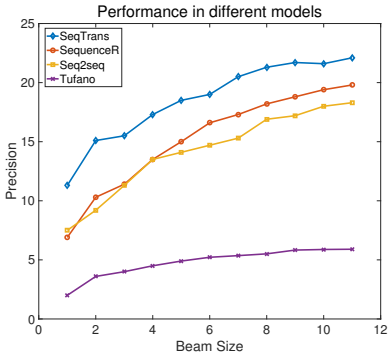


Fig. 6: Performance on D_{small}

is too long and contains a large number of declaration statements that are not related to the buggy line, which performs not well in our public vulnerable dataset. Another disadvantage is that SequenceR only supports single line prediction, but in vulnerability fixing it always contains line deletion and addition. Our approach SeqTrans supports not only statement-level code replacement but also method-level prediction which contains line deletion and addition.

In our SeqTrans, we only maintain the data dependencies before the vulnerable statement. Meanwhile, we will normalize the data and replace variable names by "$var1, var2....vark$". Literal and numerical value will also be replaced by constants and maintained in a dictionary for future recovery. The poor performance of Tufano et al. may be due to few data samples, we strictly follow their method and only select sequences with less than 100 tokens. Overall, data structure can significantly affects the performance of NMT models. Our model leverages def-use chains [20] to maintain data dependencies, which can help the NMT model reach higher accuracy.

The second experiment is multi-line prediction. Because of some implementation issues, we only compared the transformer and the seq2seq models. We input the same training set which contains a list of method-level abstracted code diffs with def-use chains. The validation process is the same as the experiment of single line prediction. Results in Table 2 shows that our transformer model achieves an accuracy of 491/8036 (6.1%) when Beam=1 on D_{small}, followed by Seq2seq 453/8036 (5.6%). On D_{median}, SeqTrans also reaches the best accuracy of 4705/10047 (46.8%), followed by Seq2seq 3201/10047 (31.9%).

The gap is even larger than the result of single line prediction. We think the reason is that if we utilize the whole vulnerable functions as the training set, the token number is always bigger than 1000, which may be too long for seq2seq model to capture relationships between each token.

Answer to RQ1: In summary, NMT models are able to learn meaningful code changes from historical code repair records and generate predicted code like a developer. Our approach SeqTrans based on transformer model outperforms other NMT model in both single line prediction and multi-line prediction. Even outperforms the state-of-the-art approach SequenceR in our public vulnerability fix dataset.

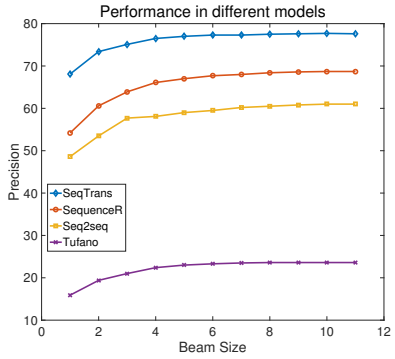


Fig. 7: Performance on D_{median}

TABLE 1: Performance results of different criteria in single line prediction

Approach	Beam	Accuracy	
		D_{small}	D_{median}
SeqTrans	1	242/2130(11.3%)	2270/3334(68.1%)
	10	338/2130(15.5%)	2504/3334(75.1%)
	50	473/2130(22.1%)	2587/3334(77.6%)
SequenceR	1	252/3661(6.9%)	2498/4610(54.2%)
	10	418/3661(11.4%)	2946/4610(63.90%)
	50	725/3661(19.8%)	3167/4610(68.7%)
Seq2seq	1	121/2130(7.5%)	1621/3334(48.6%)
	10	242/2130(11.3%)	1927/3334(57.7%)
	50	390/2130(18.3%)	2032/3330(61.0%)
Tufano et al.	1	20/1010(2.0%)	252/1577(15.9%)
	10	41/1010(4.0%)	335/1577(21.0%)
	50	63/1010(5.9%)	373/1577(23.6%)

TABLE 2: Performance results of different criteria in multiple line prediction

Approach	Beam	Accuracy	
		D_{small}	D_{median}
SeqTrans	1	491/8036(6.1%)	4705/10047(46.8%)
	10	1176/8036(14.6%)	4862/10047(48.4%)
	50	1531/8036(19.1%)	5275/10047(52.5%)
Seq2seq	1	453/8036(5.6%)	3201/10047(31.9%)
	10	1289/8036(15.9%)	4377/10047(43.6%)
	50	1320/8036(16.4%)	443/10047(44.5%)

4.3.2 RQ2: Does SeqTrans perform better in predicting specific types of CWEs?

We now look at what types of vulnerabilities fix our model can well identify and generate predictions. Figure 8, Figure 9 and Table 3 shows the prediction accuracy of each CWE. The Common Weakness Enumeration (CWE) is a category system for software weaknesses and vulnerabilities. Every CWE contains a list of CVEs. In order to make the picture more concise, we deleted the CWE with an accuracy rate of 0. Figure 8 presents the prediction accuracy in D_{median}. As we have shown before that the average accuracy of Beam=1 is 61.6%, 8 types of CWEs have higher accuracy. The highest one is CWE-287, which achieves the accuracy of 86%. Figure 9 is the prediction accuracy in D_{small}. 13 types of CWEs is higher than the average accuracy of 11.3%. The highest one is CWE-306 and it achieves a surprising prediction performance of 50%, 5 times average performance. Detailed results are given in Table 3. *CWE No.* indicates the CWE number. The first column of *Accu* is the right prediction number and the total prediction number. The second column of *Accu* is prediction accuracy. A surprising finding is that one CWE with the highest prediction accuracy in one data set achieves 0 in another data set. We can also find out there are more CWEs in D_{small} with the prediction accuracy that is higher than the average, which may due to the fact that the distribution in D_{small} is more sparse. In the following, we will compare the difference between these two datasets and make a detailed analysis of why the model performs so well on certain specific CWEs.

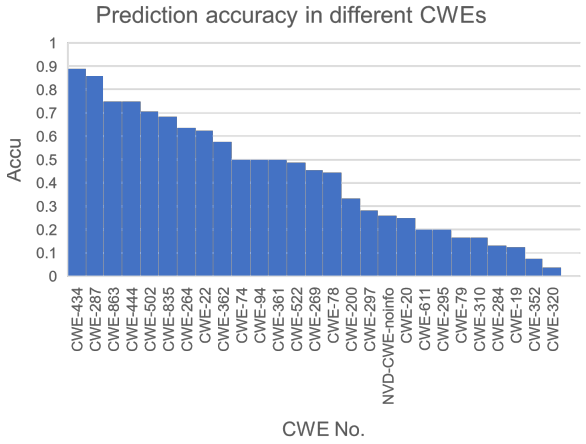


Fig. 8: Prediction accuracy of each CWE (D_{median})

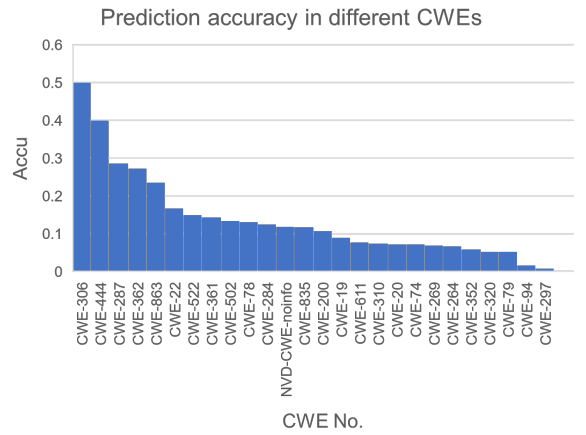


Fig. 9: Prediction accuracy of each CWE (D_{small})

In order to deeply analyze these specific CWEs, we derived Table 4 that shows the classification of code transformations by manually analyzing prediction results and source codes. We analyzed not only the correct prediction but also the wrong prediction. The first column is the type

TABLE 3: Prediction results in the data set

D_meidan			D_small		
CWE No.	**Accu**		**CWE No.**	**Accu**	
CWE-287	188/219	0.86	**CWE-306**	1/2	0.5
CWE-863	42/56	0.75	CWE-444	2/5	0.4
CWE-444	21/28	0.75	CWE-287	24/84	0.29
CWE-502	1351/1911	0.71	CWE-362	3/11	0.27
CWE-22	65/104	0.64	CWE-863	4/17	0.24
CWE-362	15/26	0.63	CWE-22	5/30	0.17
CWE-94	8/16	0.5	CWE-522	10/67	0.15
CWE-361	2/4	0.5	CWE-361	1/7	0.14
CWE-522	41/84	0.49	**CWE-502**	202/1511	0.13
CWE-78	12/27	0.44	CWE-78	3/23	0.13
CWE-200	40/120	0.33	CWE-284	1/8	0.13
CWE-297	47/166	0.28	CWE-noinfo	7/59	0.12
CWE-noinfo	18/69	0.26	CWE-200	3/28	0.11
CWE-20	28/112	0.25	CWE-19	5/56	0.09
CWE-611	28/112	0.2	CWE-611	4/52	0.08
CWE-79	7/42	0.17	CWE-310	15/202	0.07
CWE-310	27/163	0.17	CWE-20	7/97	0.07
CWE-284	10/76	0.13	CWE-74	1/14	0.07
CWE-19	1/8	0.13	CWE-269	2/29	0.07
CWE-835	1/10	0.1	CWE-264	4/60	0.07
CWE-264	5/60	0.08	CWE-352	1/17	0.06
CWE-352	3/40	0.08	CWE-320	3/57	0.06
CWE-269	2/29	0.07	CWE-79	2/37	0.05
CWE-320	1/26	0.04	**CWE-94**	1/61	0.01
CWE-74	0/14	0	CWE-297	1/140	0.01
CWE-434	0/3	0	CWE-835	0/10	0
CWE-306	0/3	0	CWE-434	0/3	0
CWE-295	0/12	0	CWE-295	0/12	0
CWE-918	0/16	0	CWE-918	0/8	0
CWE-521	0/4	0	CWE-521	0/2	0
CWE-89	0/3	0	CWE-89	0/3	0
CWE-327	0/1	0	CWE-327	0/1	0
CWE-732	0/2	0	CWE-732	0/2	0
CWE-Other	0/7	0	CWE-Other	0/6	0

name of code transformations. We roughly divided the code transformation types into 17 categories. It is worth noting that some single predictions can include multiple types of code changes, they are classified into different code change types. For this reason, the sum of the classified changes is not equalled to the number in Table 3. Detailed definitions are shown in the following:

- Change Parameter: Add, delete the parameter or change the parameter order.
- Change Throw Exception: Add, delete or replace the block of throw exception, add or delete the exception keywords in method declaration.
- Change Variable Definition: Change variable type or value.
- Change Method Call: Add, delete a method call or replace a method call by another.
- Change Target: Maintain the same method call but change the target of the method call.
- Change Annotation: Add, delete or replace the annotation.
- Change Method Declaration: Add, delete or replace method name and the qualifier.
- Change Class Declaration: Modify the declaration of a class.
- Change if Condition: Add, delete or replace operands and operators in the if condition.
- Change Switch Block: Add, delete or replace the "case" statement.
- Change Loop Condition: Modify the loop condition.
- Change Return Statement: Change return type or value, add or delete "return" keyword.
- Change Keywords "this/super": add or delete these keywords.
- Add Try Block: Put statements into the try block.
- Change Catch Exception: Add, delete or replace the block of catch exception.
- Refactoring: Rewrite the code without changing functionality.
- Other: Other transformations which are hard to be categorized or occur infrequently.

TABLE 4: Types of code transformation learned by SeqTrans

Code Transformations	**Number**	
	D_{small}	D_{median}
Change Parameter	51/495(10.3%)	535/818(65.4%)
Change Throw Exception	52/227(22.9%)	216/447(48.3%)
Change Variable Definition	24/265(9.8%)	195/418(46.7%)
Change Method Call	25/194(12.9%)	69/231(29.9%)
Change Target	14/123(11.3%)	99/221(44.8%)
Change Annotation	40/178(22.5%)	248/346(71.7%)
Change Method Declaration	28/197(14.2%)	119/229(52.0%)
Change Class Declaration	1/57(1.8%)	31/101(30.7%)
Change If Condition	10/167(6.0%)	170/292(58.2%)
Change Switch block	3/31(9.7%)	8/54(14.8%)
Change Loop Condition	2/38(5.3%)	12/41(29.3%)
Change Return Statement	5/180(2.8%)	181/453(40.0%)
Change Keywords "this/super"	6/18(33.3%)	24/41(58.5%)
Add Try Block	2/17(11.8%)	15/27(55.6%)
Change Catch Exception	1/13(7.7%)	7/36(19.4%)
Refactoring	4/85(4.7%)	89/159(56.0%)
Other	3/22(13.6%)	57/136(41.9%)

We can observe some conclusions from Table 4. In D_{small} of Table 4, SeqTrans performs well in predicting throw exception, annotation and keywords changes. All of them achieve the accuracy that twice as good as the average. When predicting method call, target, method declaration and try block changes. SeqTrans also performs better than the average accuracy. In D_{median}, SeqTrans performs well in most of the code transformations. Only method call, class declaration, switch block, loop condition, catch exception changes show lower accuracy than others. Some of them involve sophisticated code changes, while others may only be due to insufficient samples, resulting in the model not learning well.

Finding 1: SeqTrans performs well in handling throw exception change, annotation change and keywords change in both datasets. When SeqTrans is trained on D_{median}, SeqTrans can handle nearly 70% of the code transformations, all of them are higher than or close to 50% accuracy. Simple code transformations is easier to be learned by the model, even in unseen situations. Multiple line and complex code transformations may require more training data to be learned by the model.

In the following, we will discuss some CWEs in Table 3. These CWEs with significant differences in prediction performance between the two datasets have been bolded. All of them perform poorly or even achieves 0 accuracy in one dataset. We will focus on these CWEs to illustrate the differences in the same model using different training datasets.

Case Study: CWE-306: CWE-306 means "Missing Authentication for Critical Function". The software does not perform any authentication for functionality that requires a provable user identity or consumes a significant amount of resources. It is a specific case which is the only one performs better in D_{small} than in D_{median}. The right prediction (second line) is to add annotation "@SuppressWarnings ("resource")" before the method declaration.

```
> public static JMXConnectorServer createJMXServer (int port, boolean local) throws IOException
= @SuppressWarnings ( "resource" ) public static JMXConnectorServer createJMXServer (int
port, boolean local) throws IOException
< @Override public static JMXConnectorServer createJMXServer (int port, boolean local) throws
IOException
```

Fig. 10: Case: wrong prediction of CWE-835

However, as shown in the third line, the model using D_{median} incorrectly predicts the annotation, which may due to the reason that the token "@Override" appears more frequently in the training set and achieves a higher attention weight. The other two incorrect predictions belong to variable definition changes, neither model is able to make the correct prediction.

Case Study: CWE-94: CWE-94 means "Improper Control of Generation of Code". The software constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behaviour of the intended code segment. The only one case that is handled by D_{small} belongs to parameter change. The other code transformations of CWE-94 belongs to variable definition changes and catch exception change, which can be seen from Table 4 that both of them perform poorly on D_{small} but perform well in D_{median}.

```
> return URLDecoder.decode ( translatedInput, encoding)
= return URLDecoder. decode ( encoding, translatedInput)
< return URLDecoder. decode ( encoding, translatedInput)
```

Fig. 11: Case: right prediction of CWE-94

Case Study: CWE-502: CWE-502 means "Deserialization of Untrusted Data". The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid. CWE-502 related code transformations account for half of the entire training set. It contains large numbers of repetitive code transformations, such as delete one throw exception and add a return statement, change parameter orders. We will list some typical code changes that are well captured and handled by SeqTrans.

```
> throw data.instantiationException(_valueClass, ClassUtil.getRootCause(cause))
= return data.handleInstantiationProblem(_valueClass, root, ClassUtil getRootCause(cause))
< return data.handleInstantiationProblem(_valueClass, root, ClassUtil.getRootCause(cause))
```

Fig. 12: Case: right prediction of CWE-502

In Figure 12, developer delete the throw keyword and add a return keyword to transfer the instantiation problem. In addition, a new parameter was inserted into the second position. This code transformation can be well captured by SeqTrans.

```
> if (type.isAssignableFrom(raw))
= if (raw.getParameterCount( ) == 1)
< if (raw.getParameterCount( ) == 1)
```

Fig. 13: Case: right prediction of CWE-502

In Figure 13, developer firstly changes the target of the method call. Then, replace the method call from "isAssignableFrom" to "getParameterCount". Finally, the conditional expression "== 1" is added. This code transformation contains three single code transformations but is also well captured by SeqTrans. In general, our tool SeqTrans performs stable and outstandingly for vulnerability fixes like CWE-502 that contain a lot of repetitive code transformations.

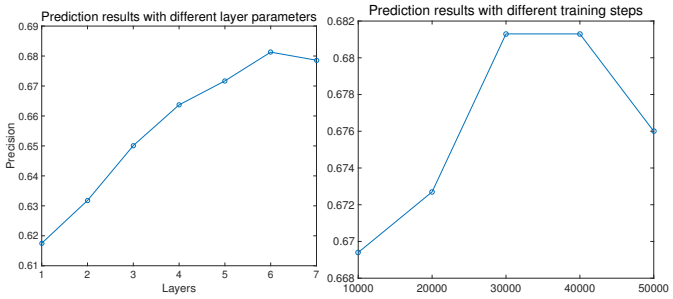
Overall, for some CWEs that contain duplicate vulnerability fixes or can be learned from historical repair records, our SeqTrans performs very well.

Finding 2: SeqTrans performs well in predicting specific kinds of vulnerability fixes like CWE-287 (Improper Authentication) and CWE-863 (Incorrect Authorization). It can well predict most code transformations in D_{median} such as annotation change and throw exception change. The prediction range will become wider and wider as the historical repair records increases.

4.3.3 RQ3: Sensitivity Analysis of SeqTrans

TABLE 5: Factor impact analysis with selected parameters

Factor	Precision	Impact
SeqTrans Model	0.6813	-
Smaller Word Size (256 vs 512)	0.6543	-4%
Larger Word Size (512 vs 1024)	0.6672	-3%
Without Data Dependency	0.5786	-15%
Without Code Normalization	0.6436	-6%
Without Copy Mechanism	0.6553	-4%



(a) Layers (b) Training Steps

Fig. 14: Factor analysis with selected parameters

Table 5 and Figure 14 shows an sensitivity analysis for each major component of SeqTrans.

From Table 5, we can see the prediction result of our SeqTrans against the results of single changes on the model. We will explain them one by one.

In the third and fourth rows we have explored the effect of word size on the performance of our model. Results shows that both the smaller and larger word size perform worse than the feature that we choose. We think the reason is that Smaller word sizes may lead to transitional compression of features and loss of some valid information. Larger word sizes may not be appropriate for the size of our dataset.

In the fifth row we have discussed whether data dependency can help the model produce better predictions. Result shows a 15% improvement in model performance when comparing our data structure to the original single vulnerable line. Result in the sixth row shows that ode normalization in data preprocessing will lead to a 6% increase in performance. The seventh row shows that the copy mechanism wo choose to mitigate OOV problem will result in a 4% increase in performance.

Figure 14 is the factor analysis for different model layers and training steps. Figure 14a is the test of model layers, we have tried different features and the conclusion is that 6 layers is a suitable choice. It is worth noting that we need to ensure that the encoder and decoder parts of the transformer model have the same number of layers, so we use the same number of layers on both encoder and decoder. Results show that prediction performance rises with the number of layers until it reaches 6. The performance of layer 7 is not better than 6, so we decide on 6 as the parameter.

Figure 14 is the experiment result for different training steps. We can see that performance rises as steps goes up until it reaches 30000. The performance of step 40000 is identical to 30000 and the performance of step 50000 is worse. This may be due to the overfitting of the model to the training data.

The sensitivity analysis results demonstrate that parameter selections for the SeqTrans produce the highest acceptance rates for the configurations we tested.

5 Threats to Validity

5.1 Internal Validity

The performance of the NMT model can be significantly influenced by the hyperparameters we adopted. The transformer model is susceptible to hyperparameters. In order to mimic the Google setup we set a bunch of options suggested by OpenNMT [39] to simulate their result. However, there are gaps between source code language and natural language. We also modified and test part of the hyperparameters and choose the one that achieves the best performance.

We manually analyzed the prediction result and the source code, classified them into 17 types. This number of categories is based on our experience during the experiment process, which may not be complete enough to cover all the code transformations. More refined classification may lead to more discoveries. However, during our analysis, we find that most of the code changes can be categorized into specific code transformations or a list of them. Only a few code changes cannot be identified, classified and even partly should be attributed to the mismatch of Gumtree [28].

5.2 External Validity

Our training data set comes from Ponta's work that published in MSR 2019 [24]. Because our goal is automatic vulnerability fix, we do not apply our tool on Defect4J [40], which is a real-world bug repair repository and is widely utilized by some works. In order to simulate the real world environment, we conduct 10 cross-validations. We believe this can also lead to the same observation. We will search for a suitable dataset to validate the performance in the future.

During the experiment we find that Gumtree [28] will introduce mismatches, which will affect the quality of the training set. In order to solve this, we fixed some bugs in Gumtree and submitted the author. We also modified Gumtree to support statement-level code matching. We believe that through these we have minimized the impact of Gumtree.

Moreover, our experiment is only based on Java language now. However, we believe that there is a common logic between programming languages, and the rules and features learned by the model can be easily applied to other languages.

6 Related Works

In recent years, Deep Learning (DL) has become a powerful tool to solve problems of Software Engineering (SE), which can capture and discover features by the DL model rather than manual derivation. In this work, we apply the Neural Machine Translation (NMT) model into the program repair field to learn from historical vulnerability repair records, summarize common pattern rules to apply to subsequent vulnerability fix. In the following, we will introduce works focus on program repair and compare our work with related research.

Program Repair Meditor [8] provides a novel algorithm that flexibly locates and groups MR (migration-related) code changes in commits. For edit application, Meditor matches a given program with inferred edits to decide which edit is applicable and produce a migrated version for developers. AppEvolve [10] can automatically perform app updates for API changes based on examples of how other developers evolved their apps for the same changes. This technique is able to update 85% of the API changes considered, but it is

quite time-consuming and not scalable enough. ARJA-e [41] proposes a new evolutionary repair system for Java code that aims to address challenges for the search space. These works are still based on statistical ranking or strict context matching. However, more and more works are beginning to exploit machine learning to rank the similar code transformations and automatically generate code recommendations.

DeepFix [42] is a program repair tool using a multi-layered sequence-to-sequence neural network with attention for fixing common programming errors. In a collection of 6,971 incorrect C language programs written by students for 93 programming tasks, DeepFix can completely repair 1881 (27%) of them, and can partially repair 1338 (19%) of them. TRACER [43] is another work that is very similar to Deepfix for fixing compiler errors, and its accuracy rate exceeds that of Deepfix. Tufano [17], [38] has investigated the feasibility of using neural machine translation for learning wild code. The disadvantage of his method is that only sentences with less than 100 tokens are analyzed.

SequenceR [18] presents a novel end-to-end approach to program repair based on sequence-to-sequence learning. It utilizes the copy mechanism to overcome the unlimited vocabulary problem. To the best of our knowledge, it achieves the best result reported on such a task. However, the abstract data structure of this method retains too much useless context. It does not use the normalization method either.

Transformer and Tree Structure Another popular direction is utilizing a transformer model or treat source code as a syntax tree to maintain richer information. TranS^3 [44] proposes a transformer-based framework to integrate code summarization with code search. Tree-based neural network such as TreeLSTM [45], [46], ASTNN [47] or TreeNet [48] are also being applied on program analysis. Shiv et al. [49] propose a method to extend transformers to tree-structured data. This approach abstracts the sinusoidal positional encodings of the transformer, using a novel positional encoding scheme to represent node positions within trees. It achieves a 22% absolute increase in accuracy on a JavaScript to CoffeeScript [50] translation dataset. TreeCaps [51] proposes a tree-based capsule network for processing program code in an automated way that encodes code syntactical structures and captures code dependencies more accurately. However, to the best-of-our knowledge, there is no work using tree-based neural machine translation for program repairing at the time of writing this paper. This situation is more challenging than translate one language to another language. Converting the generated prediction tree into readable code also faces challenges. Overall, we believe that using a tree-based neural network or even combining it with a transformer structure will become a future work of us.

7 Conclusion

In this paper, we design the automatic vulnerability fix tool SeqTrans that is based on the NMT technique to learn from historical vulnerability fixes. It can provide suggestions and automatically fix the source code for developers. We conduct our study on real-world vulnerability fix records and compare our SeqTrans with three kinds of other NMT techniques. We investigated two research questions based on these collected data. Experiment results show that our technique outperforms state-of-the-art NMT model and achieves an accuracy rate of 73.6% in single line prediction and 52.5% in multiple line prediction. We also look deeply into the model and manually analyze the prediction result and the source code. Our observation finds that SeqTrans performs quite well in specific kinds of CWEs like CWE-287(Improper Authentication) and CWE-863 (Incorrect Authorization). The prediction range will become wider and wider as the historical repair records increases.

References

[1] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[4] M. Zalewski, "American fuzzy lop: a security-oriented fuzzer," *URl: http://lcamtuf.coredump.cx/afl/(visited on 06/21/2017)*, 2010.

[5] K. Serebryany and M. Böhme, "Aflgo: Directing afl to reach specific target locations," 2017.

[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[7] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[8] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 335–346.

[9] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[10] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 204–215.

[11] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 47–50.

[12] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically assisting android api migrations using code examples," *arXiv preprint arXiv:1812.04894*, 2018.

[13] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.

[14] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: a refactoring-aware software merging technique," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[15] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 483–494.

[16] I. Eclipse, "Eclipse ide," *Website www. eclipse. org Last visited: July*, 2009.

[17] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 832–837.

[18] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[19] C. Casalnuovo, K. Sagae, and P. Devanbu, "Studying the difference between natural and programming language corpora," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1823–1868, 2019.

[20] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do i use the wrong definition? defuse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 160–174.

[21] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[23] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *arXiv preprint arXiv:1704.04368*, 2017.

[24] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[25] "Supplementary Material," https://www.dropbox.com/sh/ycwi8apjy978vmw/AABnAGE4IZMBfIIgJGclK_aWa?dl=0, 2020.

[26] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[27] "Eclipse jgit," https://www.eclipse.org/jgit/, Accessed April 4, 2017.

[28] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[29] "Scitools understand," https://scitools.com/features/, Sep 20, 2019.

[30] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[31] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.

[32] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[34] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[35] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[36] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," *arXiv preprint arXiv:1702.01806*, 2017.

[37] "Findbugs™ - find bugs in java programs," http://findbugs.sourceforge.net/, March 06, 2015.

[38] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[39] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proc. ACL*, 2017. [Online]. Available: https://doi.org/10.18653/v1/P17-4012

[40] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[41] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–53, 2020.

[42] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[43] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: for the student programs, from the student programs," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 2018, pp. 78–87.

[44] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "Transˆ 3: A transformer-based framework for unifying code summarization and code search," *arXiv preprint arXiv:2003.03238*, 2020.

[45] M. Ahmed, M. R. Samee, and R. E. Mercer, "Improving tree-lstm with tree attention," in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*. IEEE, 2019, pp. 247–254.

[46] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[47] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[48] Z. Cheng, C. Yuan, J. Li, and H. Yang, "Treenet: Learning sentence representations with unconstrained tree structure." in *IJCAI*, 2018, pp. 4005–4011.

[49] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Advances in Neural Information Processing Systems*, 2019, pp. 12 058–12 068.

[50] J. Ashkenas *et al.*, "Coffeescript," 2009.

[51] V. Jayasundara, N. D. Q. Bui, L. Jiang, and D. Lo, "Treecaps: Tree-structured capsule networks for program source code processing," *arXiv preprint arXiv:1910.12306*, 2019.